Assembler Project Description

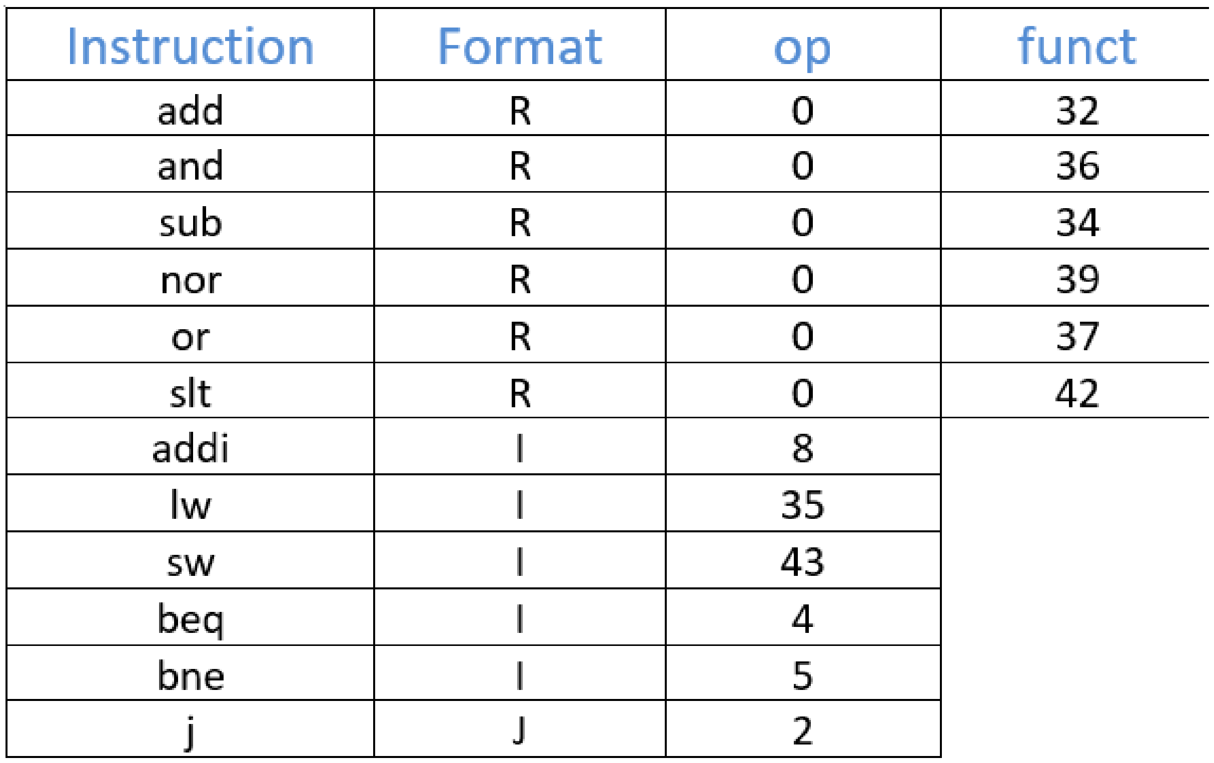
FCIS - Architecture Course – Spring 2019

1 Project Description

Write a C# program which takes a MIPS assembly program from the user and shows its corresponding machine code.The required instructions are

1. R-type instructions: add, and, sub, or, nor and slt,
2. I-type instructions: addi, lw, sw, beq, bne and
3. J-type instruction: j.

Note: op codes and funct **<u>in decimal</u>**



Instruction	Format	op	funct
add	R	0	32
and	R	0	36
sub	R	0	34
nor	R	0	39
or	R	0	37
slt	R	0	42
addi	I	8	
lw	I	35	
sw	I	43	
beq	I	4	
bne	I	5	
j	J	2	

Figure 1. Instructions

2 Instruction Conversion Examples:

Example 1: OR $t0, $zero, $zero

Binary: 00000000000000000100000000100101

Hex: 0x00004025

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL	$zero	$zero	$t0	0	OR
000000	00000	00000	01000	00000	100101
6	5	5	5	5	6

Example 2: LW $t9, 0x0038($zero)

Binary: 10001100000110010000000000111000

Hex: 0x8c190038

31 26	25 21	20 16	15 0
LW	$zero	$t9	offset
100011	00000	11001	0000000000111000
6	5	5	16

Example 3: J L1 (Assume L1 = 0x16)

Binary: 00001000000000000000000000010110

Hex: 0x08000016

31 26	25 0
J	target
000010	00000000000000000000010110
6	26

A complete program that find the first twelve Fibonacci numbers along with its translated machine codecan be found at the end of this document.

You can also use the QtSpim tool to check the machine code translation: http://spimsimulator.sourceforge.net/

3 Project Tasks

3.1 User Input

- The user code consists of data declarations followed by a program code
- The data declaration starts with an assembler directive **.data**
- The code starts with assembler directive **.text**

3.2 Project output

- Your assembler should output the binary translation of each segment in two separate textboxes or text file. Each segment contains *n* lines consisting of 32 binary digits

3.3 High-level Tasks

The project should do the following high-level tasks:

1. Scan for all labels either in the code or data segments to calculate and store their addresses
2. Allocate space for memory variables, keep their addresses and fill the initialization data
3. Translate the instructions
4. Ignore comments (anything after # symbol)

Remember that an assembler considers only the translation of the assembly code into the binary code <u>not to</u> run the instructions themselves.

3.3.1 Scan for labels

In the beginning, you need to extract all labels from input assembly code. Then, you can start translating the instructions.

1. You need to have a counter that counts the memory locations allocated for each instruction or variable. Remember that memory in MIPS is byte-addressable (each byte has an address) and all instructions are 4 bytes each. Thus, for each assembly instruction, you need to increment this counter by 4.

2. Since we have a separate memory for data and another for code, each segment starts at address 0. Thus, you should have 2 address counters (one for data segment and another for code segment)

3. Search for **labels** in both the data and code segments store them with their corresponding **address** into a generic data structure e.g. **dictionary<string, string>**.

4. A label is a string followed by a colon : (like, L1:, loop:)
5. In data segment, you need to allocate variables and fill their initial value based on the given storage type. Storage types are either .word or .space as explained later.

3.3.2 Data Declarations

Format for declarations: ***name:*** ***data_type*** ***value(s)***

- create storage for variable of the specified type with given name and specified value
- Storage types are:
  - **.word *value*** which allocates 4 bytes of memory and fill them by the given initial ***value***
  - **.space *n*** which allocates ***n*** uninitialized words (n*4 bytes).
- ***value(s)*** usually gives initial value(s); for storage type .space, gives number of spaces to be allocated

Examples:

```
var1:   .word   3        # create a single integer variable (4 bytes) with initial value 3
arr1:   .word   3, 2, 4  # create an integer array (12 bytes) with initial values 3, 2, 4
array2: .space  10       # allocate 10 consecutive words (40 bytes) (uninitialized)
```

3.3.3 Code translation

Once labels are recognized and their addresses are recorded, you can translate the assembly instructions into their binary code. You should have a hash table of instruction names and its corresponding opcode and funct and another hash table for registers and their regcodes.

- **R-type:** map instructions and registers to their corresponding binary codes. **I-Type:** translate the immediate value (in addi), the offset (in lw/sw) or the relative address of the label (in beq/bne) into their binary value.
- **J-type:** translate the direct the address of the label into binary

3.4 NOT Required Feature: Handling Errors

A user may mistype an instruction or a label. A real-world assembler should show error messages for the user, but this feature is somehow time consuming. Therefore, it is not required to handle user errors in your project. We assume that the input user code is error-free.

3.5 Programming Hints

This project depends on string manipulation and searching which is a good skill to have as a programmer. To efficiently search for patterns in a string, it is strongly recommended to use **Regular Expression** which facilitates your implementation.

- Regex Tutorial
- Using Regular Expression in C#
- Using Regular Expression in python

4 Complete Example

```
#############################################################
# This program computes the first twelve Fibonacci numbers and
# store them in array, then display them on $s0
################################################################
      .data
fibs: .space 12         # "array" of 12 words to contain fib values
size: .word  12         # size of "array"
one:  .word  1
four: .word  4
      .text
      or   $t0, $zero, $zero   # load address of array "fibs"
      lw   $t5, size ($zero)        # load array size
      lw   $t8, one ($zero)         # load one in $t8
      lw   $t9, four ($zero)        # load four in $t9
     add   $t2, $zero, $t8     # 1 is first and second Fib. number
                               # (li   $t2, 1)
      sw   $t2, 0($t0)         # F[0] = 1
      sw   $t2, 4($t0)         # F[1] = F[0] = 1
      sub $t1, $t5, $t8
      sub $t1, $t1, $t8   # Counter for loop, will execute
                               # (size-2) times
loop: lw   $t3, 0($t0)         # Get value from array F[n]
      lw   $t4, 4($t0)         # Get value from array F[n+1]
     add  $t2, $t3, $t4        # $t2 = F[n] + F[n+1]
      sw  $t2, 8($t0)          # Store F[n+2] = F[n] + F[n+1] in
                               # array
      add $t0, $t0, $t9        # increment address of Fib. number
      sub $t1, $t1, $t8        # decrement loop counter
      slt $at, $zero, $t1      # repeat if not finished yet
                               #(bgt $t1, $zero, loop)
       beq     $at, $zero, exit
       j       loop
exit: and $a0, $zero, $a0      # first argument for output (array)
      add $a1, $zero, $t5      # second argument for output (size)
     j    output               # jump to output routine.
ret:  sub $s0, $s0, $t8        # put last value of $s0 = -1
                               # ($s0  = 0 - 1)

#this routine should output 1st 12 fibonacci numbers in $s0
output: and $t0, $t0, $zero   #initialize counter (i = 0)
for: add $t1, $t0, $t0
     add $t1, $t1, $t1         #multiply i by 4
     add  $t2, $a0, $t1        #address of array element
     lw   $s0, 0($t2)          #load value of array element (F[i])
     add $t0, $t0, $t8         #i++
     slt  $at, $t0, $a1        # repeat if not finished yet.
                               # (bgt $a1, $t0, for)
     bne  $at, $zero, for
     nor  $t8, $zero, $zero    # $t8 = 1's (i.e. = -1)
     nor  $s0, $s0, $t8        # $s0 = 0
exit2:    j    ret             #return

#Translation of Data Segment
MEMORY(0) <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
MEMORY(1) <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
MEMORY(2) <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
MEMORY(3) <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
MEMORY(4) <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
MEMORY(5) <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
MEMORY(6) <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
MEMORY(7)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
MEMORY(8)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
MEMORY(9)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
MEMORY(10)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
MEMORY(11)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
MEMORY(12)  <= "00000000000000000000000000001100" ;
MEMORY(13)  <= "00000000000000000000000000000001" ;
MEMORY(14)  <= "00000000000000000000000000000100" ;

#Translation of Code Segment
MEMORY(0)  := "00000000000000001000000000100101" ;
MEMORY(1)  := "10001100000011010000000000110000" ;
MEMORY(2)  := "10001100000011000000000000110100" ;
MEMORY(3)  := "10001100000011001000000000111000" ;
MEMORY(4)  := "00000000000110000101000000100000" ;
MEMORY(5)  := "10101101000010100000000000000000" ;
MEMORY(6)  := "10101101000010100000000000000100" ;
MEMORY(7)  := "00000001101110000100100000100010" ;
MEMORY(8)  := "00000001001110000100100000100010" ;
MEMORY(9)  := "10001101000010110000000000000000" ;
MEMORY(10)  := "10001101000011000000000000000100" ;
MEMORY(11)  := "00000001011011000101000000100000" ;
MEMORY(12)  := "10101101000010100000000000001000" ;
MEMORY(13)  := "00000001000110010100000000100000" ;
MEMORY(14)  := "00000001001110000100100000100010" ;
MEMORY(15)  := "00000000000010010000100000101010" ;
MEMORY(16)  := "00010000001000000000000000000001" ;
MEMORY(17)  := "00001000000000000000000000001001" ;
MEMORY(18)  := "00000000000000100001000000100100" ;
MEMORY(19)  := "00000000000011010010100000100000" ;
MEMORY(20)  := "00001000000000000000000000010110" ;
MEMORY(21)  := "00000000000110001000000000100010" ;
MEMORY(22)  := "00000001000000000100000000100100" ;
MEMORY(23)  := "00000001000010000100100000100000" ;
MEMORY(24)  := "00000001001010010100100000100000" ;
MEMORY(25)  := "00000000100010010101000000100000" ;
MEMORY(26)  := "10001101010100000000000000000000" ;
MEMORY(27) := "00000001000110000100000000100000" ;
MEMORY(28) := "00000001000001010000100000101010" ;
MEMORY(29) := "00010100001000001111111111111001" ;
MEMORY(30) := "00000000000000001100000000100111" ;
MEMORY(31) := "00000010000110001000000000100111" ;
MEMORY(32) := "00001000000000000000000000010101" ;
```